Achieving C# .NET Interoperability with the Cascade Fast Path

PETER WU, Cornell University, Department of Computer Science, NY, USA

Currently, Cascade is an in-development distributed C++17 framework for responsive Cloud applications, especially those mindful of low latency. In particular, the edge is a primary focus, where AI/ML applications require demanding response deadlines and real-time consistency puzzles. It uses Derecho [2] under the hood, which uses remote direct memory access (RDMA) to achieve extremely fast throughput and low latency on its object store with an optimal replication mechanism based on Paxos. We focus on Cascade's "fast path", which allows logic to be injected on the data paths, and particularly, the integration of the C# .NET framework with existing C++ user-defined logic. Benchmarks on the performance of the implementation reveal that cached performance is close to native runtime, within a factor of 3 on the word count example.

Additional Key Words and Phrases: edge computing, RDMA, distributed key-value store, C++17, C#, .NET Framework, Language Integrated Query (*LINQ*), Paxos, Derecho, dynamically-linked library (*DLL*), interoperability, fast path

Peter Wu. 2023. Achieving C# .NET Interoperability with the Cascade Fast Path.

1 INTRODUCTION

The Cascade fast path [4] allows for user-defined logic (UDL) execution, and occurs analogously to the "watch" operation within a distributed hash table. It is important that this injected logic is low latency, because handlers corresponding to object pools are often run many times and must be responsive to the environment. The API offers a simple `OffCriticalDataPathObserver` (OCDPO) interface, where an implementer simply must override the function pointer to have some custom logic. This is the main entrypoint for lambda execution.

1.A C++ code for a console printer UDL; the simplest UDL possible.

```
// console_printer_udl.cpp
class ConsolePrinterOCDPO: public OffCriticalDataPathObserver {
    virtual void operator() (const node_id_t sender,
                             const std::string& key_string,
                             const uint32_t prefix_length,
                             persistent::version_t version,
                             const mutils::ByteRepresentable* const value_ptr,
                             const std::unordered_map<std::string,bool>& outputs,
                             ICascadeContext* ctxt,
                             uint32_t worker_id) override {
        std::cout << "[console printer ocdpo]: I(" << worker_id << ") received an object from sender:" <<
             sender << " with key=" << key_string << std::endl;
    }
    // ...more boilerplate to define initialization of the handler.
};
```


Manuscript submitted to ACM

These OCDPO handlers are compiled into dynamically-linked libraries (DLLs), which are specified in config files so that Cascade can find them at runtime and call their overridden function whenever necessary. This allows for succinct implementation at no expense to execution speed, and allows for modularity through easily consumable `.so` DLL files.

However, this development paradigm is not quite extensible to a lot of projects the Cascade group and its benefactors want to target: a clear example is AI/ML applications which often contain models running in Python. It would be cumbersome to have to access model weights and perform predictions in Cascade, as the implementer would have to implement the communication across the Python/C++ boundary[1]. Similarly, users of Microsoft Azure might find that C# .NET support and documentation is abundantly available, and would also want to take advantage of useful semantics like the native Language Integrated Query (LINQ), but there is no existing support for injecting C# code to the Cascade fast path. To enable this, my research is focused on developing a performant solution to connecting Cascade UDLs to be able to use C# logic.

2 IMPLEMENTATION

In this section, we summarize the bulk of engineering efforts. Most of the implementation is not done on the main Cascade repository, but rather `cascade-demos`, which has some discrete UDL examples by me and Weijia: `https://github.com/Derecho-Project/cascade-demos` (you will need access as a member of the Derecho group to view it).

2.1 Implementing C# <-> C++ Interoperability

Hosting the Common Language Runtime (CLR) is the only viable way we could implement the intended functionality, referencing infrastructure created originally by Igor Ladnik, an open source contributor on CodeProject [3]. This article has a very useful demo on connecting an arbitrary C++ unmanaged process with some managed .DLL C# code. However, it wasn't at all a simple plug-and-play to our circumstances with Cascade: we needed to integrate it with the CMake recipe used by UDLs, and contained some unnecessary boilerplate. More significantly, it required **a priori** knowledge of what DLLs it needed to work with, which is incompatible with the design of Cascade UDLs which can be provided at runtime. Upon some discussion with Professor Birman and Weijia, I opted to create a C++ UDL responsible for actually calling the C# code in the DLL. We will term this UDL the "*infrastructure DLL*". First, by hardcoding the DLL paths, I was able to get a simple console printer example working. The overridden function pointer in code block 1.A was replaced by a simple function that just creates and instance of the `GatewayToManaged` class, which has the logic for loading the CoreCLR, or runtime for .NET Core which contains the garbage collector, JIT compiler, and some primitive low level data types and classes. This is required in order to be able to load DLLs into memory on the C++ side, and was strongly inspired from Ladnik's open source example. Next, it simply invokes a managed function on the C# side responsible for invoking a specified managed direct method.

2.1.A Infrastructure DLL code for upcalling into C#

```
// csharp_udl.cpp
class CSharpOCDPO: public DefaultOffCriticalDataPathObserver {
    DLLPathModulePair dll_metadata;
```

[1]Weijia Song is the main implementer of the Python version of this effort, and we are working together.

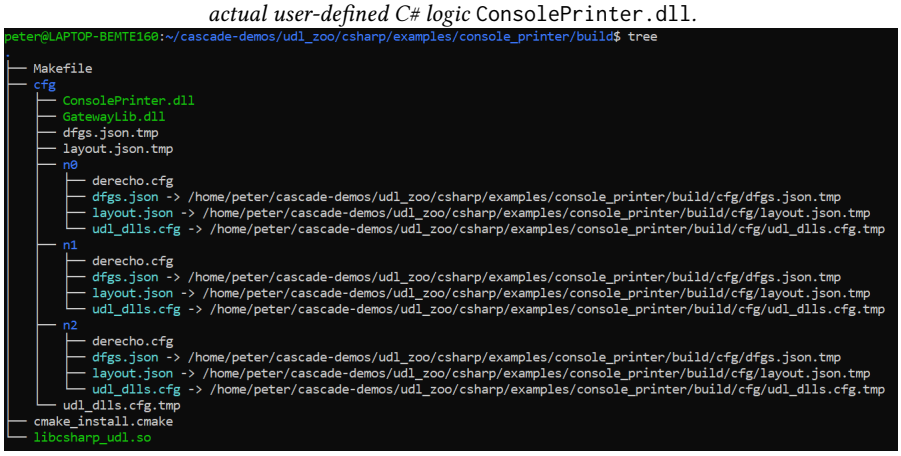
```
    virtual void ocdpo_handler (
            const node_id_t          sender,
            const std::string&       object_pool_pathname,
            const std::string&       key_string,
            const ObjectWithStringKey& object,
            const emit_func_t&       emit,
            DefaultCascadeContextType* typed_ctxt,
            uint32_t                 worker_id) override {
        std::cout << "[csharp ocdpo]: calling into managed code from sender=" << sender <<
            " with key=" << key_string << std::endl;
        gateway->Invoke(dll_metadata.dll_absolute_path.c_str(), dll_metadata.module_name.c_str(),
            {sender, object_pool_pathname.c_str(), key_string.c_str(), object.key.c_str(),
            object.blob.bytes, object.blob.bytes_size(), worker_id, &emit},
            [](const emit_func_t* emit_ptr, const char* key, const uint8_t* bytes, const uint32_t size) {
                Blob blob_wrapper(bytes, size, true);
                (*emit_ptr)(std::string(key), EMIT_NO_VERSION_AND_TIMESTAMP, blob_wrapper);
            });
    }

/* ---- static members follow ---- */
private:
    /* singleton attributes */
    static std::atomic<bool> gateway_is_initialized;
    static std::mutex     gateway_initialization_mutex;
    static std::unique_ptr<GatewayToManaged> gateway;
    static ServiceClientAPI& capi;
}
```

Note the mutex we placed on the `GatewayToManaged` singleton, as well as the atomic flag. Since UDLs can be created concurrently through their `initialize()` functions, we need to make sure that the CoreCLR runtime is only started up once, and this acts as an entryway to all managed method calls, called through reflection. We will elaborate on the reasoning behind the reflection design below.

2.1.1 Using reflection to handle managed function calls. As noted before, Ladnik's code requires knowledge of exactly which DLLs it needs to use beforehand while the CoreCLR initializes itself. However, we only initialize the CoreCLR through the `GatewayToManaged` instance once, as it is a singleton static variable shared across all OCDPO instances. Thus, this issue with .NET hosting led us to consider other avenues to be able to integrate with the Cascade system.

A key insight is that we know the DLL relative paths from the `dfgs.json` object pool configuration file, which lists UDLs associated with certain object pools. Using these relative paths, we can call methods of other C# DLLs on-the-fly just by knowing where they are, and provided they have the same method signature as a specified interface. We package the C# side into a *reflection logic DLL*, which I named `GatewayLib.dll`. This was compiled with Mono, the Linux open-source C# compiler. Now, we can call any number of arbitrary C# DLLs without need for locking the runtime (as with the Python UDL), provided they implement the interface for the `OcdpoHandler`.

2.1.B The anatomy of the simplest C# Cascade UDL. Note that there is a `GatewayLib.dll` *for reflection logic as well as the actual user-defined C# logic* `ConsolePrinter.dll`.



2.2 Zero-Copy Solution for Translating Arguments and Callbacks Across the Boundary

Starting up the CoreCLR and being able to call an arbitrary static void function on the managed C# side is relatively trivial, so the majority of effort went into being able to actually send arguments over the boundary, especially efficiently, avoiding copies. Ladnik's CodeProject article makes use of JSON arguments to pass data over from unmanaged to managed, since data structures have different memory representations on both sides. While this makes it easy, this is clearly not a performant solution, because of the overhead of deserializing and serializing on top of copying even further. Understanding how to develop a zero-copy solution to this required a deep-dive into .NET Core's interop capabilities.

2.2.1 Blittable Types. To avoid as much copying as possible, we express as many fields as possible provided to each UDL handler as a set of blittable types [1]. These types do not require any *marshalling*, or conversion which requires a copying overhead to translate the data type, for example from a C++ `std::string` to a C# `System.String`. These types include `System.Byte`, `System.Int32`, `System.UInt64` which we use as a conversion for `std::size_t` and most importantly, the opaque pointer `System.IntPtr`. The rest of the blittable types are enumerated at Microsoft's .NET documentation.

2.2.A Code for the OcdpoArgs struct and OcdpoHandler, which receives the struct on the C# side

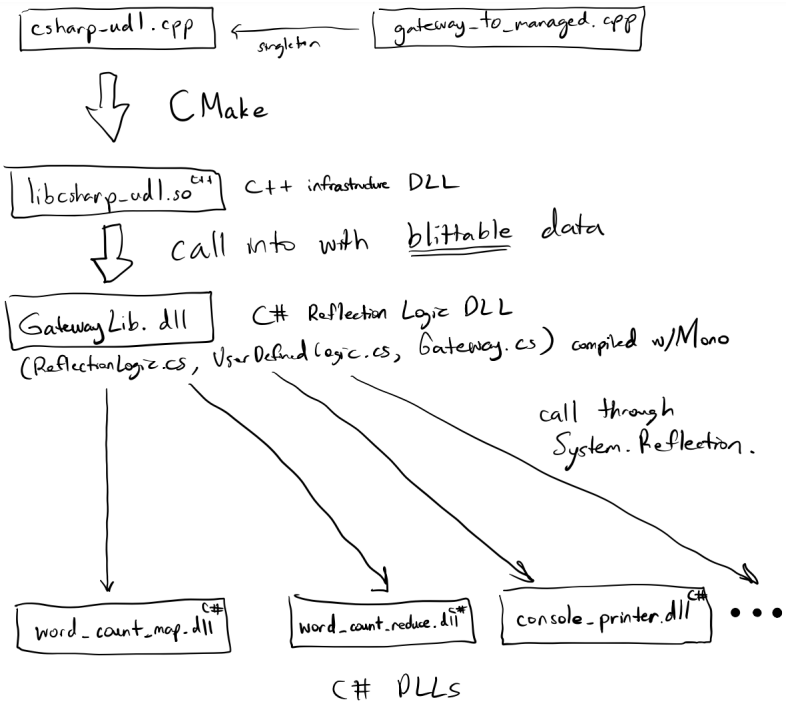
```
// gateway_to_managed.hpp
typedef struct {
   node_id_t sender;
   const char* const object_pool_pathname;
   const char* const key_string;
   const char* const object_key;
   const uint8_t* object_bytes;
   std::size_t object_bytes_size;
   uint32_t worker_id;
   const derecho::cascade::emit_func_t* emit_func;
} OcdpoArgs;
```

```
// AnyArbitraryUDL.cs
public unsafe static void OcdpoHandler(
    UInt32 sender,
    IntPtr objectPoolPathname,
    IntPtr keyString,
    IntPtr objectKey,
    IntPtr objectBytes,
    UInt64 objectBytesSize,
    UInt32 workerId,
    IntPtr emitFunc,
    IntPtr emitInvokePtr)
{
// UDL code... Note that all of the above are blittable, so no copying is required
}
```

Copying is still unavoidable in the cases where types vary between C++ and C#, such as the small overhead for string conversion when we need to access the key string or object pool pathname. We do this only when necessary, learning from the mistakes of CORBA in class, and do lazy deserialization when needed.

Interestingly, we can even avoid a copy for the biggest bottleneck—the object bytes. As seen in code 2.1.A, we the object blob bytes as a byte pointer, which the C# code can reinterpret for themselves. This prevents any UDL applications from running exceptionally slowly in C#.

2.2.B Architecture of our reflection-based solution.



2.3 Memory Safety Guarantees and a Case Study: Word Count UDL

We skip the Console Printer UDL implementation (the "Hello World" of UDLs) due to its extreme brevity, which only requires 3 lines of code. The word count example provides a more holistic introduction of how the C# UDL API works.

2.3.A Code for the WordCountMapperUDL

```
// WordCountMapper.cs
string text = Marshal.PtrToStringAnsi(objectBytes, (Int32) objectBytesSize);
Console.WriteLine("[word count mapper]: Text: " + text);
Dictionary<string, Int32> wordCount = new Dictionary<string, Int32>();
foreach (string word in text.Split(' '))
{
    if (wordCount.ContainsKey(word))
    {
        wordCount[word]++;
    }
    else
    {
        wordCount[word] = 1;
    }
}
foreach (string word in wordCount.Keys)
{
    IntPtr emitKeyPtr = Marshal.StringToHGlobalAnsi(word);
    IntPtr emitBytesPtr = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(Int32)));
    try
    {
        Int32 val = 1;
        Marshal.StructureToPtr(val, emitBytesPtr, false);
        InvokeEmit emit =
        (InvokeEmit) Marshal.GetDelegateForFunctionPointer(emitInvokePtr, typeof(InvokeEmit));
        emit(emitFunc, emitKeyPtr, emitBytesPtr, 1);
    }
    catch (Exception)
    {
        Console.WriteLine("[word_count_mapper]: Exception caught when emitting data.");
    }
    finally
    {
        // Avoid memory leaks by freeing all memory allocated here.
        Marshal.FreeHGlobal(emitKeyPtr);
        Marshal.FreeHGlobal(emitBytesPtr);
    }
}
```

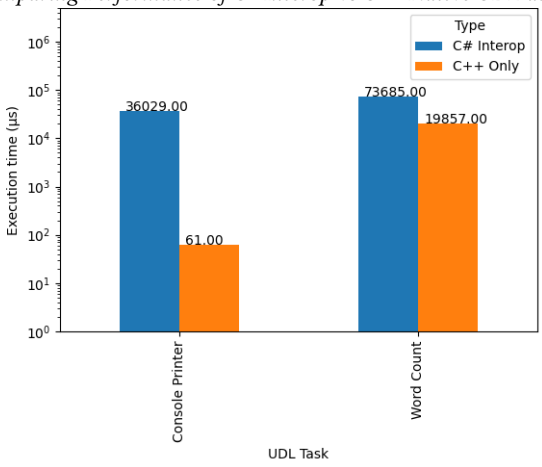
While this code is a little more involved than the C++ word count mapper, the bulk of the structure is the same: first decoding the bytes to a string in order to access the words to count, doing the computation on the string, and then emitting the data to the reducer which handles it similarly. A key point to note is the `try...catch` structure of the code, which ensures memory safety. Consider if the code crashes or reaches an exception after the memory is allocated to either the key or bytes pointers, which are to be passed back to the C++ emit callback. Then, we will never have deallocated memory, and resulted in a memory leak. The `finally` block ensures that we don't forget to free the memory after they are used in the `emit` function. We have the guarantee that these pointers will never be used again, since the Derecho `Blob` class copies the data from the byte pointers.

3 ANALYSIS

We initially hypothesized that the C# interop UDL code would clearly run slower than natively writing UDLs in C++, but mainly due to the overhead of having to translate types and copy data. In order to test this, we set up an experiment as outlined below.

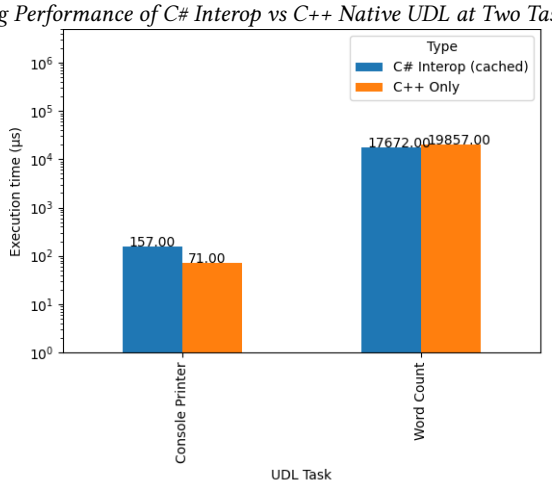
We ran the console printer and word count examples and compared the execution time from the start of the `ocdpo_handler` call to the end of execution. Specifically, for word count, which involves multiple UDLs and object pools, we take the total time as the start of the first UDL (mapper) to the end of the `ocdpo_handler` in the third UDL (reporter). We benchmarked the performance using the `std::chrono` library in microseconds, and recorded the first run time. The results of a cursory first run are shown below:

3.A Comparing Performance of C# Interop vs C++ Native UDL at Two Tasks



This first chart should be alarming—the performance of a trivial console printer execution is almost 1000-fold in C#! The use of a logarithmic scale in the y-axis was not to be sneaky, but really because the bars wouldn't be visible otherwise. Clearly, there is a significant overhead in the very first call to a reflection-based solution, as well as having to upcall through the singleton CoreCLR runtime for all handlers. However, while this overhead is not to be ignored, caching the `MethodInfo` objects after a C# is called once already results in much faster performance, as seen below:

3.B Comparing Performance of C# Interop vs C++ Native UDL at Two Tasks (Post-Caching)



Once the methods to invoke are already called once or twice, the runtime gets a lot faster: close to native runtime, within a factor of 3! The word count example actually ran even faster in C# after an initial run from figure 3.A. This eases our fears that translating the C# arguments actually causes issues due to some latent marshalling or copying, but really the biggest bottleneck is simply having to locate the method info to be invoked by reflection. Perhaps, users can warm-start their UDLs and expect close to native performance in .NET afterward.

Optimistically, our initial hypothesis was not supported by the data, especially after observing the caching effect. This is promising for performance on really large blobs for ML data and beyond, but work could still be done on the initial push.

4 FURTHER WORK / REFLECTION

Some further work this effort leaves:

- Porting over the rest of the `ServiceClientAPI` to the C# side.

- Make the client API on the C# side easier to use without having to look at existing examples.

- Further optimizations for execution speed, especially prior to caching. This could be dangerous, if we expect low latency from the first call.

REFERENCES

[1] 2022. Blittable and non-blittable types - .NET framework. https://learn.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types

[2] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.* 36, 2, Article 4 (apr 2019), 49 pages. https://doi.org/10.1145/3302258

[3] Igor Ladnik. 2019. Hosting .NET Core Components in Unmanaged C/C++ Process in Windows and Linux. https://www.codeproject.com/Articles/1276328/Hosting-NET-Core-Components-in-Unmanaged-C-Cpluspl

[4] Weijia Song, Yuting Yang, Thompson Liu, Andrea Merlina, Thiago Garrett, Roman Vitenberg, Lorenzo Rosa, Aahil Awatramani, Zheng Wang, and Ken Birman. 2022. Cascade: An Edge Computing Platform for Real-Time Machine Intelligence. In *Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems* (Salerno, Italy) *(ApPLIED '22)*. Association for Computing Machinery, New York, NY, USA, 2–6. https://doi.org/10.1145/3524053.3542741